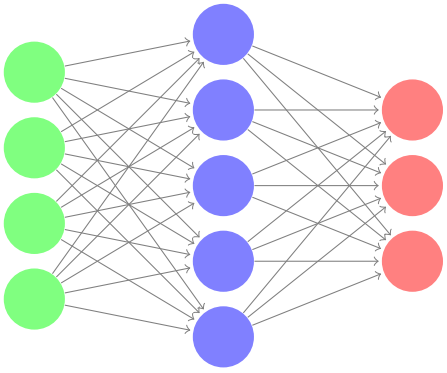
Progetto di Machine Learning Mod B

Mattia Iodice N97000284
Carlo De Vita N97000287
Francesco Romeo N86001657



Indice

Elenco delle figure

Tracce

Parte A

Progettazione ed implementazione di funzioni per simulare la propagazione in avanti di una rete neurale multi-strato con almeno: due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output.
(FACOLTATIVO: permettere all'utente di implementare reti con più di uno strato di nodi interni e con qualsiasi funzione di output per ciascun strato)
Progettazione ed implementazione di funzioni per la realizzazione della back-propagation per reti neurali multi-strato con almeno: due strati di pesi, con la sigmoide come funzione di output dei nodi interni e l'identità come funzione di output dei nodi di output, con la somma dei quadrati come funzione di errore.
(FACOLTATIVO: permettere all'utente di realizzare la back-propagation con più di uno strato di nodi interni, con qualsiasi funzione di output per ciascun strato e con qualsiasi funzione di errore derivabile rispetto all'output).

Parte B

(Difficoltà media) Si consideri come input le immagine raw del dataset mnist. Si ha, allora, un problema di classificazione a C classi, con C=10. Si estragga opportunamente un dataset globale di N coppie, e lo si divida opportunamente in training, validation e test set (ad esempio, 200 per il training set, 100 per il validation set, 100 per il test set). Si fissi la resilient backpropagation (RProp) come algoritmo di aggiornamento dei pesi (aggiornamento batch). Si studi l'apprendimento di una rete neurale con un 1, 2 e 3 strati interni di nodi confrontando il caso in cui si utilizza come funzione di output dei nodi la sigmoide con quello in cui si usa come funzione di output dei nodi la ReLU (max(0,a)). Provare diverse scelte del numero dei nodi per gli strati interni. Se è necessario, per questioni di tempi computazionali e spazio in memoria, si possono ridurre (ad esempio dimezzarle) le dimensioni delle immagini raw del dataset mnist (ad esempio utilizzando in matlab la funzione imresize).

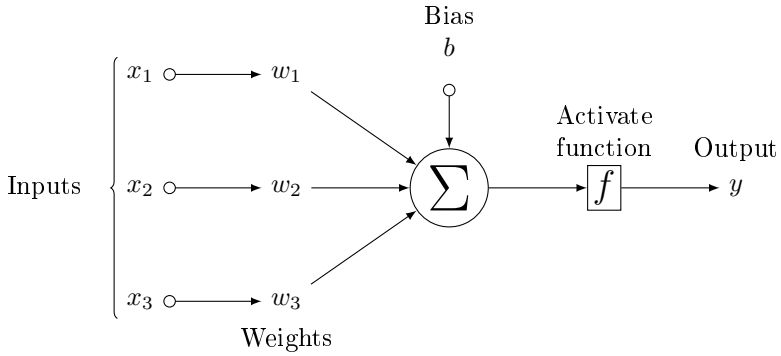


Figura 1: Neurone artificiale

1 Fase di Training

Di seguito sono riportati, in maniera molto sintetica, tutti i passi dal punto di vista teorico che definiscono la fase di **training** della rete neurale svolta sia nella Parte A del progetto che nella Parte B (in particolare ci sono leggere differenze spiegate di seguito).

1.1 Forward Propagation

La fase di **forward propagation** consiste nel "*propagare in avanti*" il calcolo delle **uscite dei nodi** di una *rete neurale feed forward multistrato* (dove per strati della rete si esclude quello di input, quindi solo quelli intermedi e quello di output). In particolare, al primo passo vengono calcolati i nodi al primo strato, grazie a questi ultimi valori è possibile calcolare quelli al secondo strato, fino a calcolare quelli all'ultimo strato (ovvero i nodi di output), ottenendo un valore per ogni nodo della rete neurale.

Di seguito è riportata la procedura per il calcolo di un'uscita di un solo nodo. Il valore di uscita a del *neurone artificiale di McCullach e Pitts* (ad esempio il neurone rappresentato in Figura 1) viene calcolato nel seguente modo:

$$a = f\left(\sum_{i=1}^{d} w_i x_i + b\right)$$

In tale funzione, w_i sono i *pesi delle connessioni* presenti tra gli ingressi e il neurone, x_i sono i valori degli *ingressi*, d è il *numero di ingressi* e b è il *bias* del neurone, ovvero un valore aggiuntivo usato nel calcolo del neurone. In particolare se si pone $b = x_0$ e $x_0 = 1$, aggiungendo una connessione w_0 tra b e il neurone, allora l'uscita del neurone può essere scritta equivalentemente nel seguente modo:

$$a = f\left(\sum_{i=0}^{d} w_i x_i\right) \tag{1}$$

È possibile osservare come tale meccanismo può essere utilizzato nel calcolo di tutte le uscite di tutti i nodi di un'intera rete neurale.

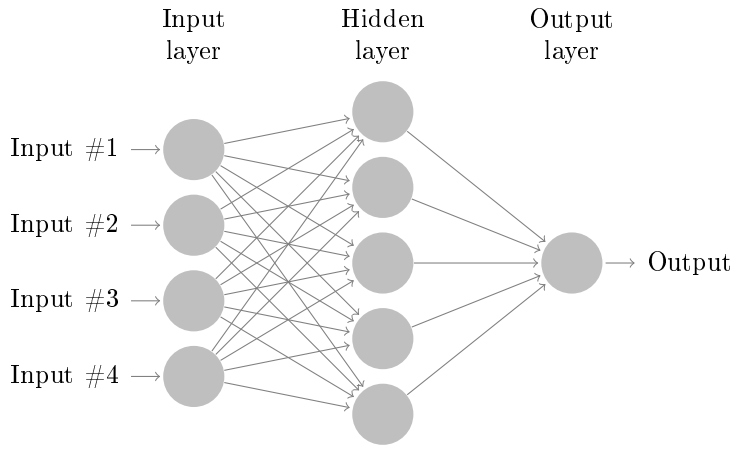


Figura 2: Rete neurale FF-ML

Iterando opportunamente la procedura, vengono così calcolati tutti i nodi di una **rete neurale feed forward multistrato** (o più brevemente *rete neurale FF-ML*) con l strati. Data una rete neurale FF-ML (ad esempio come quella mostrata in Figura 2) con le seguenti caratteristiche:

- d - dimensione del vettore di input,
- $(x_1, x_2, ..., x_d)$ - vettore di input,
- l - numero di strati,
- $(m_1, m_2, ..., m_L)$ - numero di nodi per ogni strato,
- w_{ij} - peso associato alla connessione che va dal neurone j dello, strato l-1 al neurone i dello strato l,
- f la *funzione di attivazione*;

iterando la formula del neurone artificiale, la feed forward calcola i valori di uscita per ogni nodo della rete. Per il primo strato (denominato con (1) nella formula) viene eseguita la seguente operazione che calcola tutti i nodi ${z_i}^{(1)}$ di tale strato:

$$ {z_i}^{(1)} = f\left(\sum_{j=1}^{d} {w_{ij}}^{(1)} x_j + {b_i}^{(1)}\right) $$

Per il secondo strato (denominato con (2) nella formula) viene eseguita la seguente operazione che calcola tutti i nodi ${z_i}^{(2)}$ di tale strato:

$$ {z_i}^{(2)} = f\left(\sum_{j=1}^{m_1} {w_{ij}}^{(2)} {z_j}^{(1)} + {b_i}^{(2)}\right) $$

In generale, per un generico strato l (denominato con (l) nella formula) si ha che la seguente operazione che calcola tutti i nodi ${z_i}^{(l)}$ dello strato:

$$ {z_i}^{(l)} = f\left(\sum_{j=1}^{m_{l-1}} {w_{ij}}^{(l)} {z_j}^{(l-1)} + {b_i}^{(l)}\right) \tag{2} $$

Vengono calcolate tutte le uscite dei nodi della rete neurale applicando la stessa formula "*in avanti*", seguendo l'ordine topologico stabilito dalle connessioni (da qui il nome *propagazione in avanti* o *forward propagation*).

1.1.1 Funzioni di attivazione

Sono state utilizzate due diverse funzioni di attivazione, ovvero la **sigmoide** e la **ReLU**.

Sigmoide La funzione della sigmoide è la seguente:

$$ s(x) = \frac{1}{1 + e^{-x}} \tag{3} $$

Possiamo notare come la derivata di tale funzione sia estremamente semplice, questo è uno dei motivi per cui rende la sigmoide una funzione molto utilizzata nell'ambito delle reti neurali:

$$ s'(x) = s(x) * (1 - s(x)) \tag{4} $$

ReLU La funzione della **Rectified Linear Unit** (**ReLU**) è la seguente:

$$ relu(x) = max(0, x) \tag{5} $$

Anche in questo caso ci troviamo di fronte ad una funzione la cui derivata è abbastanza semplice:

$$ relu'(x) = \begin{cases} 0, & \text{se } x \leq 0 \\ 1, & \text{altrimenti} \end{cases} \tag{6} $$

Tale funzione è molto utilizzata nell'ambito del deep learning, poichè risolve il problema del vanishing gradient, ed inoltre poichè i costi computazionali si riducono notevolmente. Più avanti (in Sezione 6) viene effettuato un confronto tra la ReLU e la sigmoide, in cui vengono discussi tali punti, in relazione ad una serie di esperimenti effettuati.

1.2 Backward Propagation

Nel machine learning molti classificatori si basano su una procedura iterativa che si pone l'obiettivo di *minimizzare una data funzione di errore*. In questo modo, cercando di minimizzare in più passi l'errore commesso nel classificare alcuni elementi del dataset, si cerca di migliore l'algoritmo di apprendimento, affinché quest'ultimo "*sbagli*" il meno possibile nel classificare. Un approccio simile viene utilizzato nel modello delle reti neurali. Dopo la fase di forward propagation, viene calcolata una funzione di errore $E(\underline{w})$, dipendente da tutti i

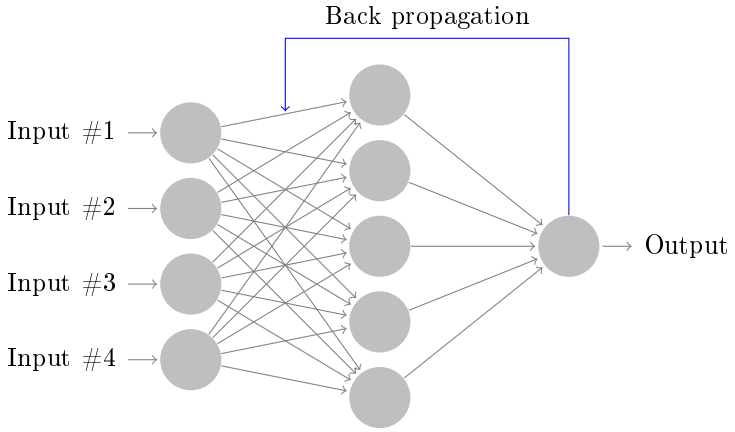


Figura 3: Rete neurale FF-ML e back propagation

pesi delle connessioni $\underline{w}$, la si cerca di minimizzare rispetto ai pesi, allo scopo di di ottenere quei pesi "*ottimali*" $w*$ indispensabili per l'apprendimento, ovvero calcolando:

$$w^* = \underset{\underline{w}}{\operatorname{argmin}} E(\underline{w})$$

Di seguito è presentata la **backward propagation** (o più semplicemente **back propagation**), una procedura che ha lo scopo di calcolare la derivata dell'errore rispetto ai pesi in maniera iterativa "*all'indietro*", a partire dall'ultimo strato (quello di output), fino ad arrivare al primo strato (come rappresentato concettualmente in Figura 3). Così facendo in una fase successiva è possibile, grazie al calcolo delle derivate, calcolare i pesi w^*. La back propagation è un metodo molto efficiente per il calcolo delle derivate della funzione di errore rispetto ai pesi di una rete. Tale procedura vale per:

- Reti neurali FF-ML
- Funzioni di attivazione derivabili
- Funzioni di errore derivabili rispetto all'output della rete

La funzione di errore E è data da tutti gli errori $E^{(n)}$, per ogni elemento n del dataset, ovvero $E = \sum_{n=1}^{N} E^{(n)}$. Di conseguenza la sua derivata è data da:

$$\frac{dE}{dw_{ij}} = \sum_{n=1}^{N} \frac{dE^{(n)}}{dw_{ij}} \tag{7}$$

In particolare, per il calcolo di tale derivata allora si vuole calcolare $\frac{dE^{(n)}}{dw_{ij}}$.
Si può dimostrare che:

$$\frac{dE^{(n)}}{dw_{ij}} = \delta_j z_i \tag{8}$$

dove $\delta_j = \frac{dE^{(n)}}{da_j}$. Il calcolo di tali delta cambia in base al tipo di nodi.

- Per i nodi di **output**

$$\delta_k = f'(a_k)\frac{dE^{(n)}}{dy_k} \tag{9}$$

- Per i nodi **interni**

$$\delta_j = f'(a_j)\sum_k w_{kj}\delta_k \tag{10}$$

Si ha quindi una formula definita **induttivamente** il cui *caso base* è dato dal calcolo dei delta per l'ultimo strato (strato di output). Risalendo "a ritroso", ovvero per il passo induttivo, vengono calcolati i delta per il primo strato, da cui il nome di *back propagation*. Una volta calcolati i delta è possibile moltiplicarli per le uscite dei nodi, come in formula 7, ottenendo le derivate rispetto ai pesi.

1.3 Aggiornamento dei pesi

Una volta ottenute le derivate della funzione di errore rispetto ai pesi $\frac{dE^{(n)}}{dw_{ij}}$, è possibile minimizzare tale funzione $E(\underline{w})$ (a più variabili) facendo variare i suoi ingressi, dati dai pesi delle connessioni.

1.3.1 Discesa del gradiente

Il seguente metodo, ovvero la **discesa del gradiente**, è sensibile ai minimi locali della funzione e permette di effettuare l'aggiornamento dei pesi desiderato. L'idea è quella di partire da un certo peso w_{ij}, aumentarlo se ci si trova alla sinistra del minimo (funzione decrescente), altrimenti diminuirlo viceversa, se si sta alla destra del minimo (funzione crescente), il comportamento desiderato è riassunto di seguito:

$$\begin{cases} \text{Incrementare il peso } w_{ij}, & \text{se } \frac{dE^{(n)}}{dw_{ij}} < 0 \\ \text{Decrementare il peso } w_{ij}, & \text{se } \frac{dE^{(n)}}{dw_{ij}} > 0 \end{cases}$$

Sia η un parametro chiamato **learning rate**, tale che $0 < \eta < 1$, allora si definisce l'aggiornamento di un generico peso w_{ij} attraverso la discesa del gradiente viene effettuato attraverso la seguente assegnazione:

$$w_{ij} \leftarrow w_{ij} - \eta\left(\frac{dE^{(n)}}{dw_{ij}}\right)$$

In generale per tutti i pesi $\underline{w}$ si ha che:

$$\underline{w} \leftarrow \underline{w} - \eta\left(\nabla E\right)$$

Dove ∇E è il gradiente della funzione, ovvero l'insieme di tutte le derivate parziali della funzione di errore E rispetto ai pesi w_{ij}. Inoltre l'aggiornamento dei pesi può essere eseguito nei seguenti modi:

- **Online learning**
  I pesi w_{ij} vengono aggiornati per ogni elemento del training set
- **Batch learning**
  I pesi w_{ij} vengono aggiornati sommando i gradienti calcolati per ogni elemento del training set al termine di un'epoca

1.3.2 RProp

La **RProp**, che sta per **Resilient Backpropagation**, è un metodo utilizzato per l'apprendimento di una rete neurale, nello specifico usato per l'aggiornamento dei pesi. Si tratta di una versione euristica del metodo della discesa del gradiente. Il peso all'epoca t cambia esclusivamente in base ad un valore ${\Delta w_{ij}}^{(t)}$, quest'ultimo viene calcolato nel seguente modo:

$$\Delta w_{ij}{}^{(t)} = \begin{cases} -\Delta_{ij}{}^{(t)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} > 0 \\ +\Delta_{ij}{}^{(t)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} < 0 \\ 0, & \text{altrimenti} \end{cases} \tag{11}$$

dove $\frac{dE}{dw_{ij}}{}^{(t)}$ denota la derivata dell'errore rispetto al peso w_{ij} all'epoca t. Quindi non resta che determinare i valori di ${\Delta_{ij}}^{(t)}$. Questi valori dipendono dalla derivata dell'errore all'epoca precedente e da quella all'epoca attuale, in particolare si ottengono nel seguente modo:

$$\Delta_{ij}{}^{(t)} = \begin{cases} \eta^+ * \Delta_{ij}{}^{(t-1)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} * \frac{dE}{dw_{ij}}{}^{(t-1)} > 0 \\ \eta^- * \Delta_{ij}{}^{(t-1)}, & \text{se } \frac{dE}{dw_{ij}}{}^{(t)} * \frac{dE}{dw_{ij}}{}^{(t-1)} < 0 \\ \Delta_{ij}{}^{(t-1)}, & \text{altrimenti} \end{cases} \tag{12}$$

dove η^+ ed η^- sono degli "*amplificatori*" della variazione del peso, per questi inoltre vale $0 < \eta^- < 1 < \eta^+$ (nel paper [1] è stato verificato che euristicamente fissando $\eta^+ = 1.2$ ed $\eta^- = 0.5$ si ottengono buoni amplificatori). Inoltre devono essere fissati i valori di Δ_0 (nel paper [1] sono stati fissati a 0.1) e i valori di *upper bound* (Δ_{max}) e *lower bound* (Δ_{min}) di tali variazioni dei pesi, in particolare si ha che $\Delta_{max} = 50$ e $\Delta_{min} = 1e^{-6}$ [1].

1.4 Calcolo dell'errore

Nel progetto, nonostante sia possibile inserire qualunque funzione di errore, sono stati effettuati gli esperimenti su due funzioni di errore, dipendenti da $\underline{y}$, gli output della rete neurale, e $\underline{t}$, i target effettivi. Durante il processo di training, è necessario capire quale configurazione di rete risulta essere quella migliore. Per fare ciò viene utilizzato un insieme, chiamato **validation set**, sul quale viene calcolato l'errore rispetto ai target. La rete migliore risulta essere quella sul cui viene ottenuto l'errore minore utilizzando, appunto, il validation set. Sono mostrate di seguito le funzioni di errore implementate per un **singolo elemento** del dataset.

1.4.1 Sum of squares

Una di queste è la **sum of squares**, che è solitamente utilizzata per problemi di **regressione** ed è data da:

$$\frac{\sum_j (y_j - t_j)^2}{2} \tag{13}$$

1.4.2 Cross Entropy

Mentre l'altra funzione di errore è la **cross entropy**, la quale è solitamente utilizzata per problemi di **classificazione** ed è data da:

$$-\sum_j t_j log(s(y_j)) \tag{14}$$

Softmax Quest'ultima fa uso della funzione di **softmax** $s(\underline{y})$, la quale è una funzione che trasforma le componenti di un vettore in probabilità, in particolare applicando la softmax la somma delle componenti è pari ad 1. Essa, rispetto alle uscite, è calcolata nel seguente modo:

$$\frac{e^{y_i}}{\sum_{k=1}^{N} e^{y_k}} \tag{15}$$

In particolare, è possibile osservare che tale funzione *non è stabile numericamente* e potrebbe presentare dei casi in cui la macchina restituisce risultati indesiderati una volta applicata. Così facendo è stata adottata per gli esperimenti una variante della softmax, denominata la **softmax stabile**, che è equivalente alla prima ma stabile numericamente. La softmax stabile è ricavabile semplicemente a partire dalla softmax moltiplicando e dividendo il numeratore e il denominatore per una costante C, quindi si hanno i seguenti passi:

$$\begin{aligned}
s(y_i) &= \frac{e^{y_i}}{\sum_{k=1}^{N} e^{y_k}} \quad \text{(Per la definizione 15)} \\
&= \frac{Ce^{y_i}}{C\sum_{k=1}^{N} e^{y_k}} \quad \text{(Moltiplicando e dividendo per C)} \\
&= \frac{e^{log(C)}e^{y_i}}{e^{log(C)}\sum_{k=1}^{N} e^{y_k}} \quad \text{(Ponendo } C = e^{log(C)}\text{)} \\
&= \frac{e^{y_i+log(C)}}{\sum_{k=1}^{N} e^{y_k+log(C)}} \quad \text{(Per più proprietà degli esponenziali)}
\end{aligned}$$

Ponendo $log(C) = -max(\underline{y})$, tale versione, essendo stabile numericamente a differenza della prima, è stata implementata nel progetto. Viene effettuata tale assegnazione poiché in questo modo gli esponenti avranno valori compresi nell'intervallo $(-\infty, 0]$, evitando in questo modo di raggiungere un overflow.

1.4.3 Criterio di fermata: Generalization Loss

Nell'addestramento della rete neurale, si è deciso di utilizzare un criterio di stop. In particolare è stato utilizzato il criterio della **generalization loss**. L'addestramento viene terminato quando vale:

$$\left|\left(100 * \left(\frac{emin}{eval} - 1\right)\right)\right| > threshold \tag{16}$$

2 Fase di Testing

In tale fase viene eseguita la forward propagation su tutti gli elementi del **test set** e si controlla quale tra i nodi di output presenta il valore maggiore. La classificazione di un'immagine, da parte della rete neurale, è corretta se la classe assegnata dalla rete all'immagine corrisponde a quella a cui effettivamente l'immagine appartiene. In tale fase viene calcolata l'**accuracy** sul test set come il rapporto tra le immagini classificate correttamente e il numero di immagini.

3 Implementazione della parte facoltativa

Oltre ad aver sviluppato la parte A, sono stati sviluppati anche i relativi punti facoltativi. In particolare, la funzione che crea la rete accetta due vettori:

- VettoreStrati
  Ovvero il numero di nodi per ogni strato (compreso quello di input).
- VettoreFunzioni
  Ovvero le funzioni di attivazione per ogni strato.

In questo modo il programma può creare una rete con qualsiasi tipo di configurazione di nodi, e con qualsiasi funzione di attivazione.
Inoltre il programma può eseguire l'addestramento utilizzando qualsiasi funzione di errore. In particolare, per quanto riguarda le funzioni di attivazione e le funzioni di errore, è stata creata una apposita struttura per ognuna di esse. Tale struttura contiene un puntatore alla funzione stessa, ed un puntatore alla sua derivata. Per dettagli implementati andare alla sezione 4.

4 Subroutine

Sono riportate di seguito tutte le subroutine presenti nel progetto.

loadMNIST

Input:
- *Filename1*: Path del file che contiene le immagini MNIST.
- *Filename2*: Path del file che contiene le label MNIST.

Output:
- *Out*: Una struttura contenente due matrici, la prima di dimensione Nx784 (dove N è il numero delle immagini contenute nel file), la seconda di dimensione Nx1 (contenente le etichette).

Descrizione:
Il file delle immagini viene trasformato in una matrice in cui ogni immagine 28x28 viene rappresentata da una riga (l'immagine viene quindi vista come un array monodimensionale).

relu

Input:
- Nessun parametro.

Output:
- Struttura ReLU: Una struttura contenente due puntatori a funzione, che sono la ReLU e la derivata della ReLU.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

identità

Input:
- Nessun parametro.

Output:
- Struttura identità: Una struttura contenente due puntatori a funzione, che sono l'identità e la derivata dell'identità.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

sigmoide

Input:
- Nessun parametro.

Output: - Struttura sigmoide: Una struttura contenente due puntatori a funzione, che sono la sigmoide e la derivata della sigmoide.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

softmax

Input:
- X: parametro della funzione.
Output:
- Softmax(x): l'input a cui è stata applicata la funzione di softmax stabile.

Descrizione:
La softmax stabile viene calcolata come

createNetwork

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato della rete.
- *vettoreFunzioni*: vettore contenente le funzioni di attivazioni per ogni strato di neuroni.
- *Pesi*: un numero reale tramite il quale si definisce l'intervallo di appartenenza dei pesi della rete.

Output:
- *Net*: Una struttura contenente tutte le proprietà relative alla rete: i pesi per ogni collegamento tra nodi, i bias per ogni nodo, il numero di strati interni della rete, le funzioni di attivazione per ogni strato di nodi, l'output della rete.

Descrizione:
Questa funzione crea una rete feed-forward full connected multi-layers in base ai parametri passati in input (numero di strati, funzioni di attivazione, inizializzazione dei pesi).

CrossEntropy

Input:
- Nessun parametro

Output:
- Struttura crossEntropy: una struttura contenente due puntatori a funzione, i quali puntano alla funzione e alla derivata della crossEntropy. Tale struttura, inoltre, contiene un altro campo, ovvero un puntatore alla funzione che deve essere applicata all'ultimo strato durante la forward propagation, cioè la funzione identità.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

SumOfSquares

Input:
- Nessun parametro

Output:
- Struttura crossEntropy: una struttura contenente due puntatori a funzione, i quali puntano alla funzione e alla derivata della SumOfSquares. Tale struttura, inoltre, contiene un altro campo, ovvero un puntatore alla funzione che deve essere applicata all'ultimo strato durante la forward propagation, cioè la softmax.

Descrizione:
Tale funzione ritorna la struttura sopra descritta.

forwardPropagation

Input:
- *net*: rete neurale creata con la funzione create Network(...).
- *input*: una matrice contenente le immagini del training set (viste come vettore).
- *FunErr*: è la funzione di errore utilizzata per l'addestramento.

Output:
- *net*: la stessa rete neurale data in ingresso in cui sono stati aggiornati gli output.
- *Z*: matrice contenente gli output della rete in base all'input.

Descrizione:
Questa funzione, partendo dagli input iniziali (immagini viste come vettori), calcola gli output di ogni strato utilizzando come funzione di attivazione quella che viene specificata all'interno della rete neurale. All'ultimo strato viene applicata una funzione, oltre all'identità, che dipende dal tipo di funzione di errore. Nel caso della cross entropy viene applicata la softmax.

backPropagation

Input:
- *net*: rete neurale.
- *output*: output della rete.
- *targets*: valori attesi degli output.
- *funErr*: struttura della funzione di errore che si vuole usare (ricordiamo che la struttura contiene sia la funzione di errore che la sua derivata).

Output:
- *delta*: un cell array nel quale nella posizione i-esima vi sono i delta calcolati per lo strato i-esimo.

Descrizione:
La funzione parte con il calcolare i delta dell'ultimo strato di nodi, per poi utilizzarli nel calcolo dei delta degli strati superiori.

calcolaDerivate

Input:
- *net*: rete neurale.
- *delta*: delta dei nodi della rete.
- *Input*: immagini di input (viste come vettori).

Output:
- *derW*: un cell array in cui nella posizione i-esima vi sono le derivate dei pesi dello strato i-esimo.
- *derB*: un cell array in cui nella posizione i-esima vi sono le derivate dei bias dello strato i-esimo.

Descrizione:
Vengono calcolate e derivate dei pesi e dei bias tramite i valori dei delta (dati in ingresso alla funzione) e dei valori dei nodi della rete.

aggiornaPesi

Input:
- *net*: rete neurale.
- *derW*: derivate dei pesi.
- *derB*: derivate dei bias.
- *Eta*: rappresenta il learning rate.

Output:
- *net*: rete in cui vi sono i pesi aggiornati.

Descrizione:
La funzione aggiorna tutti i pesi della rete, utilizzando il metodo della discesa

del gradiente, tenendo conto del learning rate dato in ingresso (parametro eta).

initVariazioni

Input:
- *net*: rete neurale.
- *var*: variazione.

Output:
- *varW*: un cell array in cui nella posizione vi sono le variazioni dei pesi per ogni strato.
- *varB*: un cell array in cui nella posizione vi sono le variazioni dei bias per ogni strato.

Descrizione:
Le variazioni vengono inizializzate tramite il parametro var dato in ingresso. Sia le variazioni dei pesi che dei bias vengono inizializzate allo stesso modo.

rProp

Input:
- *net*: rete neurale.
- *derW*: derivate dei pesi.
- *derB*: derovate dei bias.
- *oldDerW*: derivate dei pesi all'epoca precedente.
- *oldDerB*: derivate dei bias all'epoca precedente.
- *varW*: variazione per i pesi.
- *varB*: variazione per i bias.
- *etaP*: valore di eta+.
- *etaN*: valore di eta-.

Output:
- *net*: rete con pesi e bias aggiornati.
- *varW*: nuove variazioni calcolate per i pesi in base al segno del prodotto delle derivate oldDerW e derW.
- *varB*: nuove variazioni calcolate per i bias in base al segno del prodotto delle derivate oldDerB e derB.
- *oldDerW*: contiene le derivate dei pesi, così da utilizzarle per il calcolo dell'eta alla prossima epoca.
- *OldDerB*: contiene le derivate dei bias, così da utilizzarle per il calcolo dell'eta alla prossima epoca.

Descrizione:
La funzione calcola il segno del prodotto tra oldDerW e derW, per capire l'andamento di tale funzione. In base al segno ottenuto e ai valori etaP e etaN, aggiorna le variazioni che saranno effettuate sui pesi. Si noti che le variazioni

aggiornate sono sempre comprese nell'intervallo [1e-6, 50]. Viene eseguito lo stesso procedimento per i bias, e vengono aggiornate oldDerW e oldDerB in modo da usarle correttamente durante l'epoca successiva.

TrainingBatch

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare (che contiene la funzione d'errore e la sua derivata).
- *training Set*: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.
- *Eta*: rappresenta il learning rate.
- *etaP*: valore di eta+.
- *etaN*: valore di eta-.
- *Variation*: un reale per inizializzare le variazioni per la rprop.

Output:
- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Il tipo di aggiornamento usato per i pesi è il batch. Lo scopo di questa funzione è di restituire la migliore rete trovata durante il learning.

ParteA_ TrainingOnline

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare.
- *training Set*: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.
- *Eta*: rappresenta il learning rate.

Output:

- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).
- *soglia*: soglia per il criterio di generalization loss.

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Viene effettuato un online learning. Lo scopo di questa funzione è di restituire la migliore rete trovata durante l'addestramento.

ParteA_ TrainingRProp

Input:
- *vettoreStrati*: vettore contenente il numero di nodi per ogni strato.
- *vettoreFunzioni*: vettore contenente le funzioni da utilizzare per ogni strato.
- *Pesi*: range di valori nel quale sono contenuti i pesi.
- *funErr*: struttura della funzione di errore che si vuole utilizzare.
- *training Set*: struttura contenente sia le immagini che le labels che si vogliono usare come training set.
- *validationSet*: struttura contenente sia le immagini che le label che si vogliono usare come validation set.
- *nEpoche*: numero di epoche che si vuole utilizzare.
- *Eta*: rappresenta il learning rate.

Output:
- *bestNet*: restituisce la rete migliore trovata (in termini di funzione di errore).
- *soglia*: soglia per il criterio di generalization loss.

Descrizione:
La funzione crea una rete feed-forward full connected multi-layer, che viene addestrata utilizzando i parametri dati in input. Il tipo di aggiornamento dei pesi è il batch. Lo scopo di questa funzione è di restituire la migliore rete trovata durante l'addestramento.

testing

Input:
- *net*: la rete neurale addestrata. - *tset*: il test set sul quale calcolare l'accuracy.
Output:
- *accuracy*: Valore dell'accuratezza.
Descrizione:
Effettua il testing sulla rete neurale addestrata valutando l'accuracy sul test set.

5 Sperimentazioni con RProp

In questa sezione si fa riferimento alla parte B del progetto. In particolare vengono effettuati degli addestramenti utilizzando l'RProp come metodo di aggiornamento dei pesi. Alla prima epoca però, si è deciso di utilizzare il metodo della discesa del gradiente, in modo tale da non dover inizializzare le derivate dei pesi e dei biases.
Per quanto riguarda le sperimentazioni effettuate, è stato preso in considerazione il dataset MNIST. Sono state quindi caricate le immagini e mappate su un vettore da 784 elementi. A questo punto possiamo ben capire come le varie configurazioni della rete neurale che viene addestrata, siano tutte del tipo: 784-X1-..-Xn-10.
Sono poi stati settati diversi parametri, in particolare:
Lunghezza training set: 50000.
Lunghezza validation set: 10000.
Lunghezza test set: 10000.
η per la discesa del gradiente: 0,00001.
η^+: 1,2.
η^-: 0,5.
Delta iniziali per l'RProp: 0,001.
Inizializzazione dei pesi: compresi nell'intervallo $(-0.01, 0.01)$.

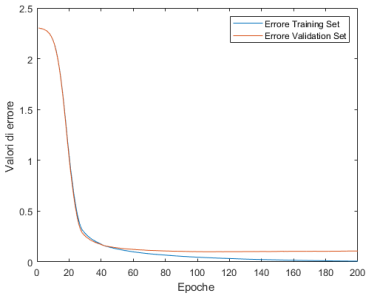
Si è deciso di inserire i risultati ottenuti dagli esperimenti in apposite tabelle, in cui sono presenti valori di accuracy e di errore minimo sul validation set. In particolare nella Tabella 1 vengono mostrati i valori di accuracy ottenuti utilizzando il test set, mentre nella Tabella 2 vengono mostrati i valori dell'errore minimo registrato sul validation set.
Come configurazioni della rete neurale sono state prese tutte le possibili combinazioni utilizzando 50 e 100 come numero di nodi per gli strati interni, considerando 1, 2 e 3 strati interni. Nella prossima pagina sono riportati i grafici delle funzioni di errore (sia dell'errore sul training che su validation) per le configurazioni discusse.

Nodi interni	Sigmoide	ReLU
50	95,68	95,6
100	95,8	96,22
200	96,31	96,63
50-50	95,1	95,32
50-100	95,14	95,61
100-50	95,51	95,73
100-100	95,54	95,62
50-50-50	93,94	93,77
50-50-100	94,2	94,69
50-100-50	94,25	94,85
50-100-100	94,29	95,08
100-50-50	94,37	94,89
100-50-100	94,3	95,51
100-100-50	95,04	95,02
100-100-100	94,41	95,75

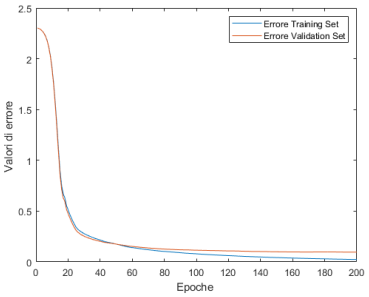
Tabella 1: Tabella con i valori di accuracy

Nodi interni	Sigmoide	ReLU
50	0,1000988	0,0964114
100	0,0934250	0,0981330
200	0,0763237	0,0778853
50-50	0,1055948	0,1025986
50-100	0,1032013	0,0974521
100-50	0,0972442	0,1012330
100-100	0,0908800	0,1054563
50-50-50	0,1337751	0,1500559
50-50-100	0,1298486	0,1212637
50-100-50	0,1314769	0,1194351
50-100-100	0,1278007	0,1146487
100-50-50	0,1228083	0,1184414
100-50-100	0,1146635	0,1146798
100-100-50	0,1018264	0,1121113
100-100-100	0,1102285	0,1051585

Tabella 2: Tabella con i valori dell'errore minimo sul validation set

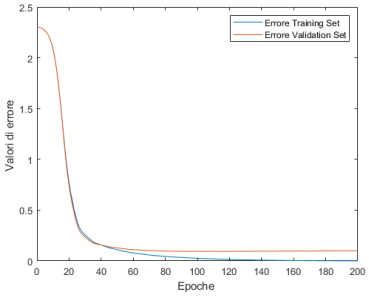


(a) *50 nodi Sigmoide*

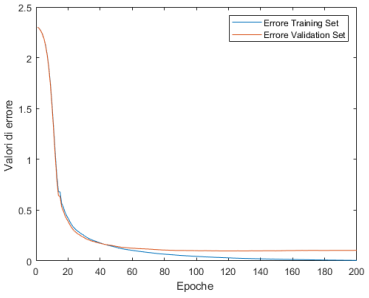


(b) *50 nodi ReLU*

Figura 4: 50 nodi confronto Sigmoide e ReLU

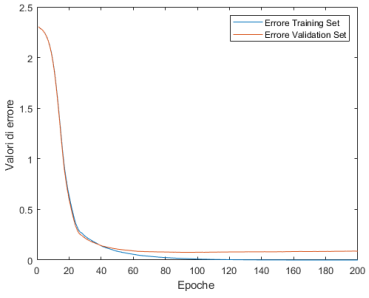


(a) *100 nodi Sigmoide*

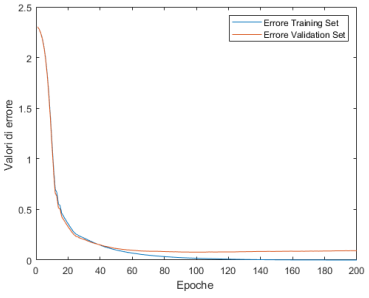


(b) *100 nodi ReLU*

Figura 5: 100 nodi confronto Sigmoide e ReLU

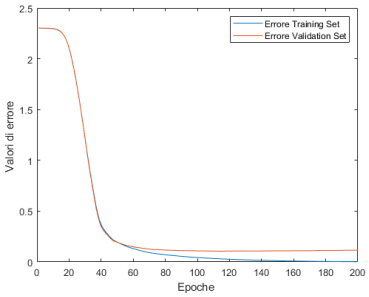


(a) *200 nodi Sigmoide*

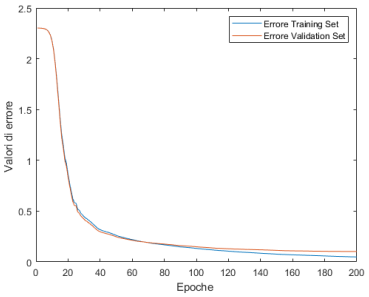


(b) *200 nodi ReLU*

Figura 6: 200 nodi confronto Sigmoide e ReLU

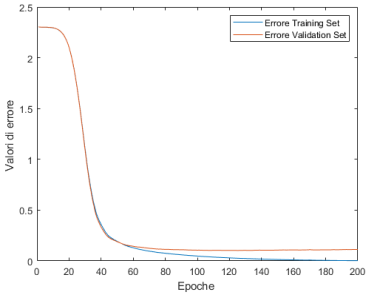


(a) *50-50 nodi Sigmoide*

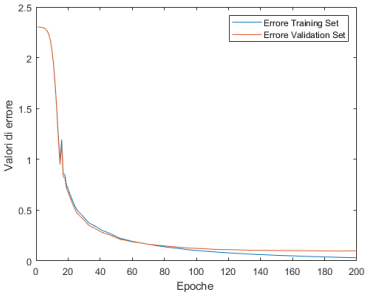


(b) *50-50 nodi ReLU*

Figura 7: 50-50 nodi confronto Sigmoide e ReLU

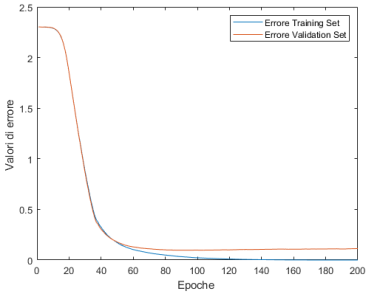


(a) *50-100 nodi Sigmoide*

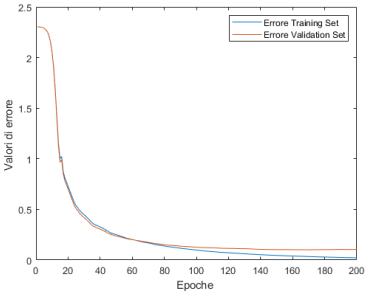


(b) *50-100 nodi ReLU*

Figura 8: 50-100 nodi confronto Sigmoide e ReLU

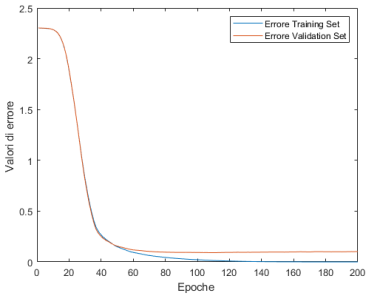


(a) *100-50 nodi Sigmoide*

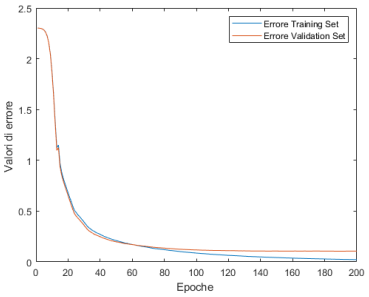


(b) *100-50 nodi ReLU*

Figura 9: 100-50 nodi confronto Sigmoide e ReLU

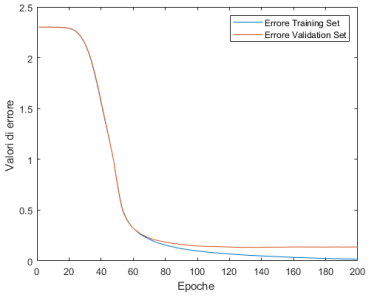


(a) *100-100 nodi Sigmoide*

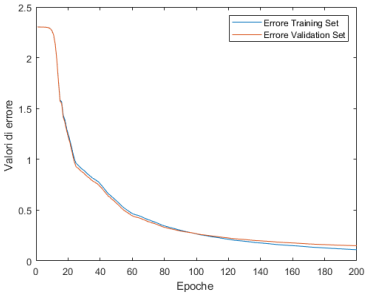


(b) *100-100 nodi ReLU*

Figura 10: 100-100 nodi confronto Sigmoide e ReLU

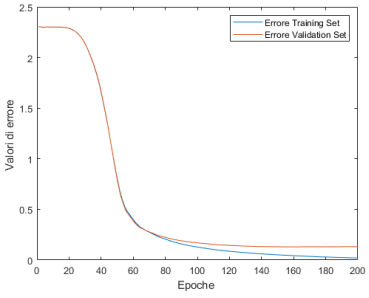


(a) *50-50-50 nodi Sigmoide*

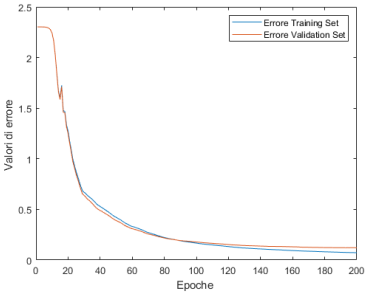


(b) *50-50-50 nodi ReLU*

Figura 11: 50-50-50 nodi confronto Sigmoide e ReLU

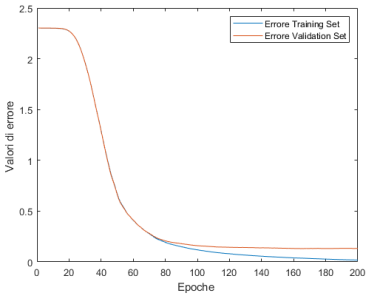


(a) *50-50-100 nodi Sigmoide*

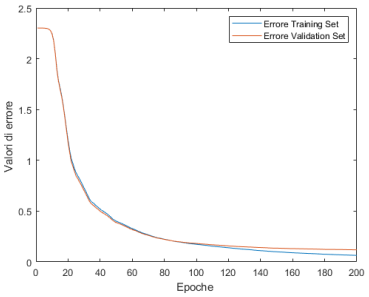


(b) *50-50-50 nodi ReLU*

Figura 12: 50-50-100 nodi confronto Sigmoide e ReLU

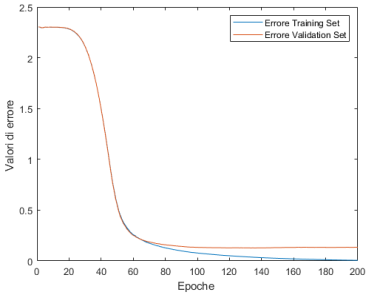


(a) *50-100-50 nodi Sigmoide*

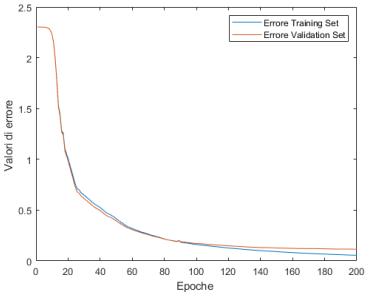


(b) *50-100-50 nodi ReLU*

Figura 13: 50-100-50 nodi confronto Sigmoide e ReLU

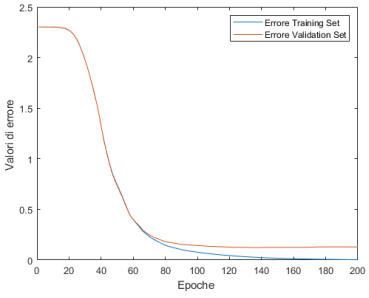


(a) *50-100-100 nodi Sigmoide*

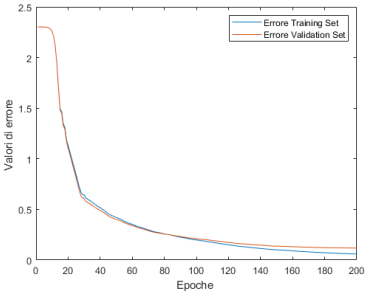


(b) *50-100-100 nodi ReLU*

Figura 14: 50-100-100 nodi confronto Sigmoide e ReLU

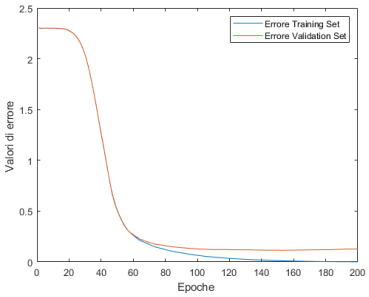


(a) *100-50-50 nodi Sigmoide*

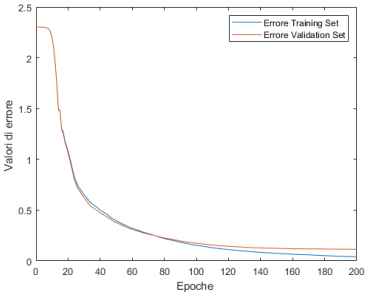


(b) *100-50-50 nodi ReLU*

Figura 15: 100-50-50 nodi confronto Sigmoide e ReLU

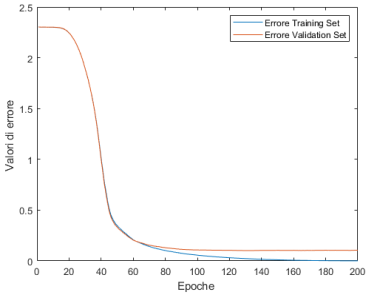


(a) *100-50-100 nodi Sigmoide*

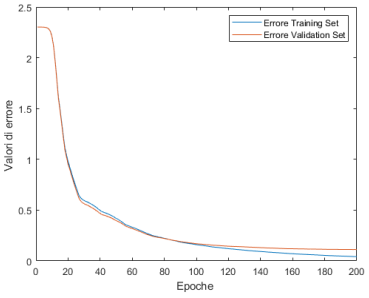


(b) *100-50-100 nodi ReLU*

Figura 16: 100-50-100 nodi confronto Sigmoide e ReLU

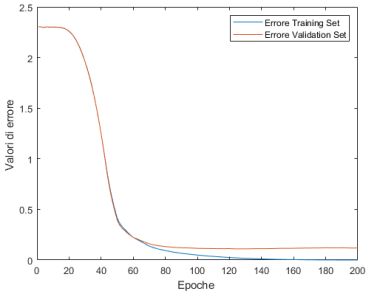


(a) *100-100-50 nodi Sigmoide*

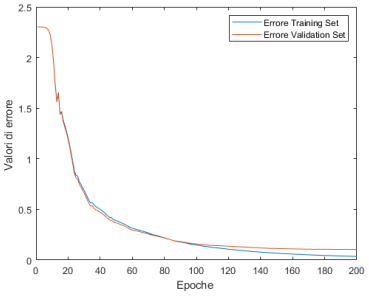


(b) *100-100-50 nodi ReLU*

Figura 17: 100-100-50 nodi confronto Sigmoide e ReLU



(a) *100-100-100 nodi Sigmoide*



(b) *100-100-100 nodi ReLU*

Figura 18: 100-100-100 nodi confronto Sigmoide e ReLU

6 Conclusioni sulle sperimentazioni

È possibile innanzitutto notare che i risultati ottenuti, su tutte le diverse configurazioni di una rete neurale, sono molto positivi. Infatti notiamo che in quasi tutti i casi l'accuracy si aggira intorno ai valori del 94% e 95%. Dalle prove eseguite quindi viene confermato che l'aggiornamento dei pesi tramite **RProp** funziona in maniera estremamente positiva. Infatti, non solo vengono raggiunti valori di errore molto bassi, ma ciò avviene in un numero di epoche relativamente piccolo. Appurata dunque la bontà della RProp, è stato effettuato un confronto tra i risultati ottenuti utilizzando la sigmoide e quelli ottenuti utilizzando la ReLU. Tali risultati sono positivi sia utilizzando l'una che l'altra funzione di attivazione, anche se la ReLU per molte configurazioni si è comportata generalmente meglio, soprattutto all'aumentare del numero di strati. Tale risultato è dovuto al fenomeno del **vanishing gradient**. Infatti con più strati si ottiene una funzione di errore che presenta *molti minimi relativi*, cioè punti in cui il gradiente è pari a 0. Questo ovviamente può creare problemi per il raggiungimento del minimo assoluto. Infatti, la backward propagation calcola il gradiente utilizzando la chain rule, ed essendo la derivata della sigmoide compresa tra i valori $(0, 0.25)$, vengono effettuati molti prodotti tra numeri molto piccoli. Se si tiene conto che i valori ottenuti su uno strato, influiscono sui valori che si otterranno allo strato precedente, possiamo ben capire come si arrivi alla presenza di valori molto piccoli del gradiente, in particolare via via che si risale durante la backward propagation. Di conseguenza, se i gradienti sono prossimi a 0, vuol dire che l'aggiornamento dei pesi sarà pressochè impercettibile. Se guardiamo il grafico della sigmoide (rappresentato in Figura 19) possiamo notare che, se

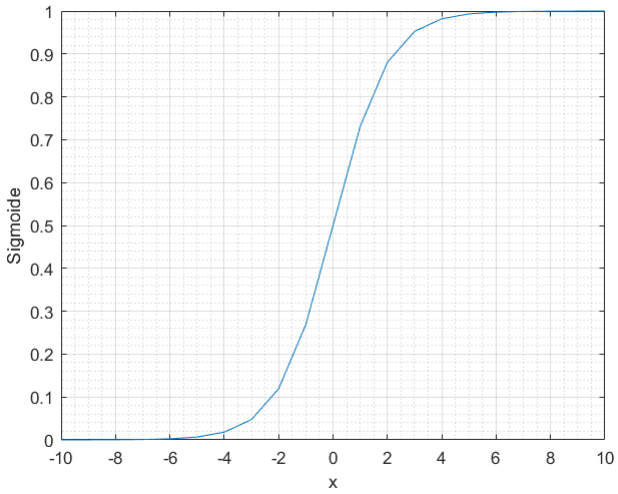


Figura 19: Sigmoide

l'input di un nodo non è vicino allo 0, la sigmoide tende ad assomigliare ad una funzione costante. Di conseguenza la derivata della sigmoide in quel punto è prossima allo 0, ma quindi anche il delta corrispondente a quel nodo è prossimo allo 0. Visto che i delta influiscono per il calcolo delle derivate, allora anche le

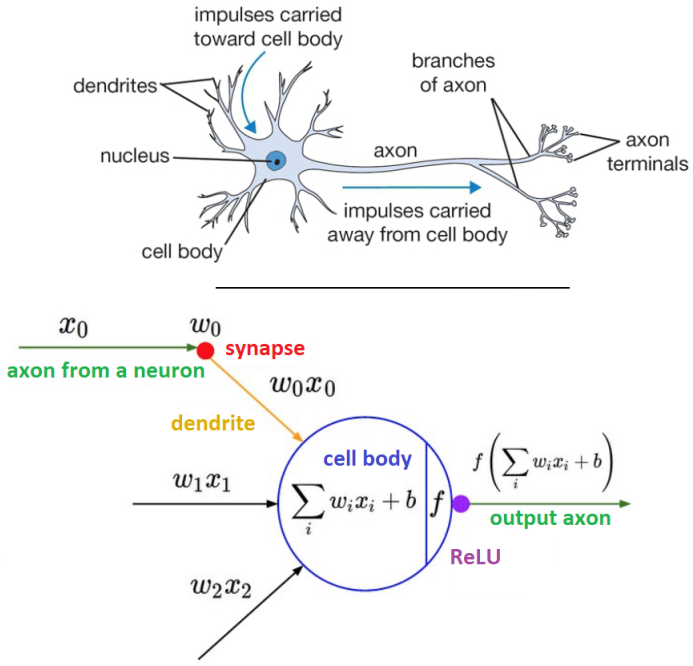


Figura 20: Neurone biologico ed implementazione attraverso ReLU

derivate dei pesi degli archi che vanno in tale nodo sono prossime allo 0. I delta inoltre, come detto precedentemente, verranno utilizzati anche per il calcolo dei delta dello strato precedente. Ma ciò vuol dire che anche i delta dello strato precedente saranno prossimi allo 0, e di conseguenza lo saranno le derivate calcolate utilizzando questi delta. Dunque possiamo notare come ci sia un aumento dei minimi locali. Proprio per evitare il proliferare di minimi locali, nelle reti neurali deep viene utilizzata la **Rectified Linear Unit** (**ReLU**). Tale funzione è definita come $f(x) = max(0, x)$ (rappresentata in Figura 21). La ReLU non soffre del problema del vanishing gradient, infatti per i nodi i cui input sono positivi, la derivata della ReLU è 1, e ciò garantisce che il gradiente non arrivi a valori molto vicini allo 0 durante la fase di back propopagation. Notiamo infatti dai grafici ottenuti dalle sperimentazioni che la ReLU converge verso l'errore minimo più velocemente di quanto lo fa la sigmoide. Con la ReLU inoltre non tutti i nodi partecipano al calcolo dell'output, pochè i nodi il cui valore di attivazione risulta minore di 0 vengono "*disattivati*", cioè valgono 0 e quindi non influiscono in alcun modo sui valori di input dei nodi dello strato successivo. In questo modo viene rappresentata quella che è la vera relazione che intercorre tra i nodi di input e quelli di output. Ciò ci fa pensare alla rete neurale biologica, infatti in essa è presente un'infinità di neuroni, ma solo alcuni di questi vengono attivati in base agli impulsi esterni (si vedi Figura 20). La sparsità conferisce maggiore potere predittivo alla rete e garantisce maggiore robustezza sui piccoli cambiamenti dell'input, poichè l'insieme di nodi attivi resterà pressochè lo stesso. La sparsità inoltre, come possiamo notare dai grafici ottenuti, permette

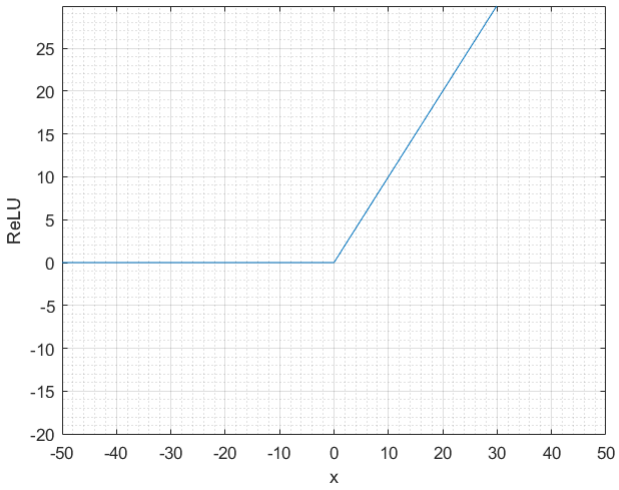


Figura 21: ReLU

anche di avere un overfitting minore. Un ulteriore vantaggio della ReLU è che i costi computazionali vengono notevolmente ridotti, poichè i calcoli da eseguire sono molto semplici. Sono stati infatti presi i tempi computazionali su un addestramento di una rete neurale 784-100-100-100-10 utilizzando 50000 immagini di training e 10000 di validation e sono stati ottenuti i seguenti valori: con la sigmoide il training ha avuto durata pari a 559,469 secondi mentre con la ReLU una durata pari a 424,295 secondi. La ReLU però può anche presentare degli svantaggi, in particolare quando ci sono troppi nodi disattivi. Ciò ovviamente comporta una perdita di potenza predittiva da parte della rete, che si manifesta in maniera totale quando tutti i neuroni sono disattivati (*Dying ReLU*). A tal proposito sono state proposte diverse soluzioni per risolvere questo problema, una tra le più appetibili risulta essere quella di scegliere come funzione di attivazione la **Leaky ReLU** (**LReLU**). La LReLU (rappresentata in Figura 22) è definita nel seguente modo:

$$\begin{cases} x, & \text{se } x \geq 0 \\ \alpha x, & \text{altrimenti} \end{cases} \tag{17}$$

con α solitamente piccolo. Tale funzione permette di dare una certa importanza anche a quei neuroni che con la ReLU sarebbero stati disattivi. Tuttavia è stato osservato empiricamente che la LReLU non è generalmente migliore della ReLU, bensì converge verso l'errore minimo più velocemente.
È stato infatti effettuato un esperimento con la LReLU, ponendo $\alpha = 0.01$ su una rete 784-100-100-100-10, ed è stata ottenuta la curva rappresentata in Figura 23. L'accuracy della rete è del 95,9% e l'errore minimo registrato sul validation set è: 0.0979305. Notiamo dal grafico dunque, come effetivamente tale funzione di errore converga più velocemente al minimo, rispetto a quella ottenuta dall'addestramento utilizzando la ReLU su una rete con la stessa configurazione di nodi e strati interni.

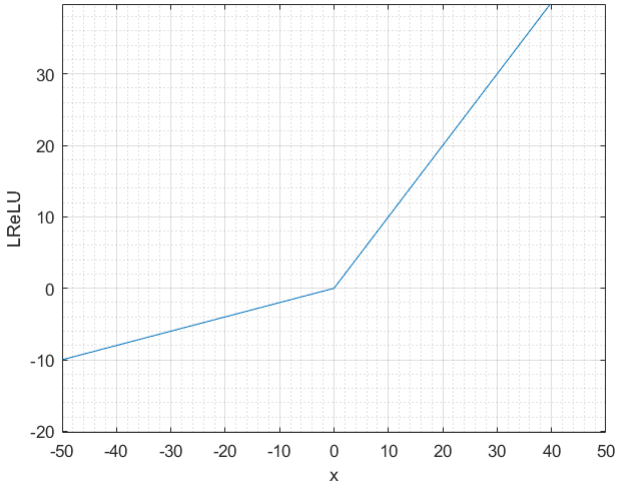


Figura 22: LReLU

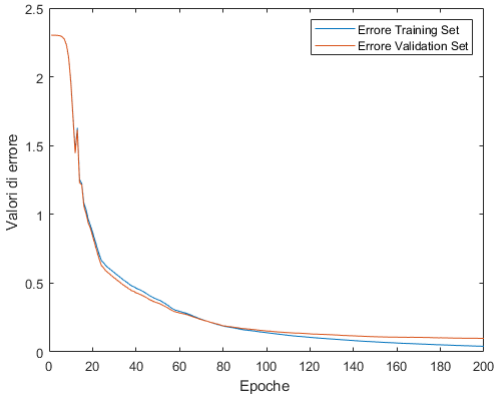


Figura 23: Esempio di utilizzo della LReLU

7 Codice Matlab

```
function net = aggiornaPesi(net,derW,derB,eta)
    %Discesa del gradiente
    for strato=1 : net.nStrati
        net.w{strato} = net.w{strato} - eta*derW{strato};
        net.b{strato} = net.b{strato} - eta*derB{strato};
    end
end
```

```
function [net,delta] = backPropagation(net,output,targets
    ,funErr)
    %Inizializzazione di un cell array per i delta.
    delta = cell(1,net.nStrati);
    %Calcolo dei delta all'ultimo strato
    delta{net.nStrati} = funErr.der(output,targets);
    %Calcolo dei delta per i strati interni
    for strato = (net.nStrati-1) : -1 : 1
        delta{strato} = delta{strato+1} * net.w{strato+1};
        delta{strato} = net.funzioni{strato}.der(net.
            output{strato}) .* delta{strato};
    end
end
```

```
function [derW,derB] = calcolaDerivate(net,delta,input)
    %Inizializzazione di cell array per le derivate dei
        pesi e dei bias
    derW = cell(1,net.nStrati);
    derB = cell(1,net.nStrati);
    %Calcolo delle derivate per tutti i pesi
    z = input;
    for strato=1 : net.nStrati
        derW{strato} = delta{strato}' *z;
        derB{strato} = sum(delta{strato},1);
        z = net.output{strato};
    end
end
```

```
function net = createNetwork(vettoreStrati,
    vettoreFunzioni,pesi)
    nStrati = length(vettoreStrati) - 1;
    w = cell(1,nStrati);
    b = cell(1,nStrati);
    dim1 = vettoreStrati(1);
    for strato = 1:nStrati
```

```
        dim2 = vettoreStrati(strato+1);
        w{strato} = pesi - (2*pesi)*rand(dim2,dim1);
        b{strato} = pesi - (2*pesi)*rand(1,dim2);
        dim1=dim2;
    end
    net.w=w;
    net.b=b;
    net.nStrati = nStrati;
    net.funzioni = vettoreFunzioni;
    net.output = cell(1,nStrati);

end
```

```
%Funzione che ritorna la struttura della crossEntropy
function e = crossEntropy()
    e.fun = @f;
    e.der = @d;
    e.lastLayerFun = @lastLayerFun;
end

%Funzione della cross entropy
function y = f(out,target)
    y = -sum(sum(target .* log(max(out,0.09)),2));
end

%Derivata della cross entropy
function y = d(out,target)
    y = (out-target);
end

%Funzione da applicare all'ultimo strato della rete
    durante la forward
%propagation
function y = lastLayerFun(x)
    y = softmax(x);
end
```

```
function [net,z] = forwardPropagation(net,input,funErr)
    z = input;
    %Cardinalita' degli elementi di input
    c = size(z,1);
    %Propagazione in avanti
    for strato= 1:net.nStrati
        a = (z * net.w{strato}') + repmat(net.b{strato},c
            ,1);
        z = net.funzioni{strato}.fun(a);
        net.output{strato}=z;
```

```
10      end
11      %Applicazione di una ulteriore funzione sull'ultimo
            strato. Nel caso
12      %della cross-entropy si tratta della softmax.
13      net.output{net.nStrati} = funErr.lastLayerFun(net.
            output{net.nStrati});
14      z = net.output{net.nStrati};
15  end
```

```
1   %Funzione che ritorna la struttura dell'identita'
2   function s = identita()
3       s.fun=@f;
4       s.der=@d;
5   end
6
7   %Funzione identita'
8   function y = f(x)
9       y=x;
10  end
11
12  %Derivata dell'identita'
13  function y = d(x)
14      y=ones(size(x));
15  end
```

```
1   function [varW,varB] = initVariazioni(net,var)
2       %Inizializzazione dei cell array
3       varW = cell(1,net.nStrati);
4       varB = cell(1,net.nStrati);
5       %Per ogni strato si crea una matrice con tutti valori
            uguali a var.
6       for strato=1:net.nStrati
7           varW{strato} = var*ones(size(net.w{strato}));
8           varB{strato} = var*ones(size(net.b{strato}));
9       end
10
11  end
```

```
1   function out = loadMNIST(filename1,filename2)
2           %Caricamento del dataset MNIST
3       fp = fopen(filename1, 'rb');
4       assert(fp ~= -1, ['Could not open ', filename1, '']);
5
6       magic = fread(fp, 1, 'int32', 0, 'ieee-be');
7       assert(magic == 2051, ['Bad magic number in ',
            filename1, '']);
```

```
8
9       numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
10      numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
11      numCols = fread(fp, 1, 'int32', 0, 'ieee-be');
12
13      images = fread(fp, inf, 'unsigned char');
14      images = reshape(images, numCols, numRows, numImages)
            ;
15      images = permute(images,[2 1 3]);
16
17      fclose(fp);
18
19      % Reshape to #pixels x #examples
20      images = reshape(images, size(images, 1) * size(
            images, 2), size(images, 3));
21      % Convert to double and rescale to [0,1]
22       images = double(images) / 255;
23
24      fp = fopen(filename2, 'rb');
25      assert(fp ~= -1, ['Could not open ', filename2, '']);
26
27      magic = fread(fp, 1, 'int32', 0, 'ieee-be');
28      assert(magic == 2049, ['Bad magic number in ',
            filename2, '']);
29
30      numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');
31
32      labels = fread(fp, inf, 'unsigned char');
33
34      assert(size(labels,1) == numLabels, 'Mismatch in
            label count');
35
36      fclose(fp);
37
38      out.Images = images';
39      out.Labels = labels;
40  end
```

```
1   function s = lrelu()
2       s.fun = @f;
3       s.der = @d;
4   end
5
6   function y = f(x)
7       y = (x>=0).*x + (x<0).*(0.2*x);
8   end
9
10  function y=d(x)
```

```
11      y = (x>0).*1 + (x==0).*0 + (x<0).*(0.2);
12  end
```

```
1   function bestNet = ParteA_trainingBatch( vettoreStrati,
        vettoreFunzioni,pesi,funErr,trainingSet,validationSet,
        nEpoche,eta,soglia)
2       %Creazione di una rete neurale
3       net = createNetwork(vettoreStrati, vettoreFunzioni,
            pesi);
4       %Array degli errori per training set e validation set
5       errT = zeros(1,nEpoche);
6       errV = zeros(1,nEpoche);
7       errMin = realmax;
8       %Processo di addestramento
9       e=1;
10      stop=0;
11      while(e<=nEpoche && stop==0)
12          [net,output] = forwardPropagation(net,trainingSet
                .images,funErr);
13          [net,delta] = backPropagation(net,output,
                trainingSet.targets,funErr);
14          [derW,derB] = calcolaDerivate(net,delta,
                trainingSet.images);
15          [net] = aggiornaPesi(net,derW,derB,eta);
16          %Calcolo dell'errore su training set
17          [net,out]=forwardPropagation(net,trainingSet.
                images,funErr);
18          errT(e) = funErr.fun(out,trainingSet.targets)/
                size(trainingSet.targets,1);
19          %Calcolo dell'errore su validation set
20          [net,out]=forwardPropagation(net,validationSet.
                images,funErr);
21          errV(e) = funErr.fun(out,validationSet.targets)/
                size(validationSet.targets,1);
22          %Se l'errore all'epoca corrente e' minore del
                minimo ottenuto, la
23          %rete corrente diventa la migliore.
24          if errV(e) < errMin
25              errMin = errV(e);
26              bestNet = net;
27          end
28          %Stampa degli errori
29          fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e)
                ,errV(e));
30          %Criterio di fermata
31          t = abs(100*((errMin/errV(e))-1));
32          if(t > soglia)
33              stop = 1;
```

```
34          end
35          e=e+1;
36
37      end
38      %Stampa errore minimo
39      fprintf("Errore minimo validation: %.7f\n",errMin);
40      figure
41      x=1:nEpoche;
42      plot(x,errT(x),x,errV(x));
43      xlabel('Epoche ') % x-axis label
44      ylabel('Valori di errore ') % y-axis label
45      legend('Errore Training Set','Errore Validation Set')
46  end
```

```
1   function bestNet = ParteA_trainingOnline( vettoreStrati,
        vettoreFunzioni,pesi,funErr,trainingSet,validationSet,
        nEpoche,eta,soglia)
2       %Creazione di una rete neurale
3       net = createNetwork(vettoreStrati, vettoreFunzioni,
            pesi);
4       %Array degli errori per training set e validation set
5       errT = zeros(1,nEpoche);
6       errV = zeros(1,nEpoche);
7       errMin = realmax;
8       %Processo di addestramento
9       e=1;
10      stop=0;
11      while(e<=nEpoche && stop==0)
12          for i=1:size(trainingSet.images,1)
13              [net,output] = forwardPropagation(net,
                    trainingSet.images(i,:),funErr);
14              [net,delta] = backPropagation(net,output,
                    trainingSet.targets(i,:),funErr);
15              [derW,derB] = calcolaDerivate(net,delta,
                    trainingSet.images(i,:));
16              [net] = aggiornaPesi(net,derW,derB,eta);
17          end
18          %Calcolo dell'errore su training set
19          [net,out]=forwardPropagation(net,trainingSet.
                images,funErr);
20          errT(e) = funErr.fun(out,trainingSet.targets)/
                size(trainingSet.targets,1);
21          %Calcolo dell'errore su validation set
22          [net,out]=forwardPropagation(net,validationSet.
                images,funErr);
23          errV(e) = funErr.fun(out,validationSet.targets)/
                size(validationSet.targets,1);
24          %Se l'errore all'epoca corrente e' minore del
```

```
            minimo ottenuto, la
25          %rete corrente diventa la migliore.
26          if errV(e) < errMin
27              errMin = errV(e);
28              bestNet = net;
29          end
30          %Stampa degli errori
31          fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e)
                ,errV(e));
32          %Criterio di fermata
33          t = abs(100*((errMin/errV(e))-1));
34          if(t > soglia)
35              stop = 1;
36          end
37          e=e+1;
38      end
39      %Stampa errore minimo
40      fprintf("Errore minimo validation: %.7f\n",errMin);
41      figure
42      x=1:nEpoche;
43      plot(x,errT(x),x,errV(x));
44      xlabel('Epoche ') % x-axis label
45      ylabel('Valori di errore ') % y-axis label
46      legend('Errore Training Set','Errore Validation Set')
47  end
```

```
1   %Funzione che ritorna la struttura della relu
2   function s = relu()
3       s.fun = @f;
4       s.der = @d;
5   end
6
7   %Funzione della relu
8   function y = f(x)
9       y = max(0,x);
10  end
11
12  %Derivata della relu
13  function y=d(x)
14      y = (x>0);
15  end
```

```
1   function [net,oldDerW,oldDerB,varW,varB] = rprop(net,derW
        ,derB,oldDerW,oldDerB,varW,varB,etaP,etaN)
2       for strato=1:net.nStrati
3           %Segno del prodotto tra le nuove e le vecchie
                derivate.
```

```
4           signProd = sign(oldDerW{strato} .* derW{strato});
5           %Individuazione degli eta da utilizzare nell'
                aggiornamento dei pesi
6           etaM = ((signProd>0) * etaP) + ((signProd<0) *
                etaN) + ((signProd==0) * 1);
7           %Calcolo del delta che rappresenta la variazione
                del peso
8           varW{strato} = etaM .* varW{strato};
9           %I delta sono compresi tra 1e-6 e 50
10          varW{strato} = min(max(varW{strato},0.000001),50);
11          %Aggiornamento dei pesi
12          net.w{strato} = net.w{strato} - (sign(derW{strato
                }) .* varW{strato});
13
14          %Segno del prodotto tra le nuove e le vecchie
                derivate.
15          signProd = sign(oldDerB{strato} .* derB{strato});
16          %Individuazione degli eta da utilizzare nell'
                aggiornamento dei bias
17          etaM = ((signProd>0) * etaP) + ((signProd<0) *
                etaN) + ((signProd==0) * 1);
18          %Calcolo del delta che rappresenta la variazione
                del bias
19          varB{strato} = etaM .* varB{strato};
20          %I delta sono compresi tra 1e-6 e 50
21          varB{strato} = min(max(varB{strato},0.000001),50);
22          %Aggiornamento dei bias
23          net.b{strato} = net.b{strato} - (sign(derB{strato
                }) .* varB{strato});
24
25      end
26      oldDerW = derW;
27      oldDerB = derB;
28  end
```

```
1   %Funzione che ritorna la struttura della sigmoide
2   function s = sigmoide()
3       s.fun=@f;
4       s.der=@d;
5   end
6
7   %Funzione della sigmoide
8   function y = f(x)
9       y = 1 ./ (1+exp(-x));
10  end
11
12  %Derivata della sigmoide
13  function y = d(x)
```

```
    y = x .* (1-x);
end
```

```
function s = softmax(x)
%     s = exp(x-max(x')') ./ sum(exp(x-max(x')'),2);
    m = max(x,[],2);
    s = exp(x-m) ./ sum(exp(x-m),2);
end
```

```
%Funzione che ritorna la struttura della 'somma dei
    quadrati'
function e = sumOfSquares()
    e.fun = @f;
    e.der = @d;
    e.lastLayerFun = @lastLayerFun;
end

%Funzione della 'somma dei quadrati'
function y= f(out,t)
    y = sum(sum((out-t).^2,2) /2);
end

%Derivata della 'somma dei quadrati'
function z = d(y,t)
    z = y-t;
end

%Funzione da applicare all'ultimo strato della rete
    durante la forward
%propagation
function y = lastLayerFun(x)
    y = x;
end
```

```
function acc = testing(net,testSet,funErr)
    acc=0;
    for i=1:size(testSet.Images,1)
        net = forwardPropagation(net,testSet.Images(i,:),
            funErr);
        [val,k]=max(net.output{net.nStrati});
        %fprintf("BEST: immagine: %d, MAX: %.7f, res: %d,
            label: %d\n",i,max(net.output{net.nStrati}),k
            ,test.Labels(i,1));
        if(k==10)
            k=0;
```

```
9           end
10          if( k == testSet.Labels(i,1))
11              acc=acc+1;
12          end
13      end
14      acc = acc/size(testSet.Images,1);
15  end
```

```
1   function bestNet = trainingRProp( vettoreStrati,
        vettoreFunzioni,pesi,funErr,trainingSet,validationSet,
        nEpoche,eta,etaP,etaN,variation,soglia)
2       %Creazione di una rete neurale
3       net = createNetwork(vettoreStrati, vettoreFunzioni,
            pesi);
4       %Inizializzazione dei delta per l'aggiornamento dei
            pesi della rProp
5       [varW,varB] = initVariazioni(net,variation);
6       %Array degli errori per training set e validation set
7       errT = zeros(1,nEpoche);
8       errV = zeros(1,nEpoche);
9       errMin = realmax;
10      %Processo di addestramento
11      e=1;
12      stop=0;
13      while(e<=nEpoche && stop==0)
14          [net,output] = forwardPropagation(net,trainingSet
                .images,funErr);
15          [net,delta] = backPropagation(net,output,
                trainingSet.targets,funErr);
16          [derW,derB] = calcolaDerivate(net,delta,
                trainingSet.images);
17          %Alla prima epoca viene eseguito un passo di
                aggiornamento dei pesi
18          %tramite la discesa del gradiente.
19          if(e==1)
20              [net] = aggiornaPesi(net,derW,derB,eta);
21              oldDerW=derW;
22              oldDerB=derB;
23          else
24              %Aggiornamento dei pesi tramite rprop.
25              [net,oldDerW,oldDerB,varW,varB] = rprop(net,
                    derW,derB,oldDerW,oldDerB,varW,varB,etaP,
                    etaN);
26          end
27          %Calcolo dell'errore su training set
28          [net,out]=forwardPropagation(net,trainingSet.
                images,funErr);
29          errT(e) = funErr.fun(out,trainingSet.targets)/
```

```
            size(trainingSet.targets,1);
30          %Calcolo dell'errore su validation set
31          [net,out]=forwardPropagation(net,validationSet.
                images,funErr);
32          errV(e) = funErr.fun(out,validationSet.targets)/
                size(validationSet.targets,1);
33          %Se l'errore all'epoca corrente e' minore del
                minimo ottenuto, la
34          %rete corrente diventa la migliore.
35          if errV(e) < errMin
36              errMin = errV(e);
37              bestNet = net;
38          end
39          %Stampa degli errori
40          fprintf("e: %d errT: %.7f errV: %.7f\n",e,errT(e)
                ,errV(e));
41          %Criterio di fermata
42          t = abs(100*((errMin/errV(e))-1));
43          if(t > soglia)
44              stop = 1;
45          end
46          e=e+1;
47      end
48      %Stampa errore minimo
49      fprintf("Errore minimo validation: %.7f\n",errMin);
50      figure
51      x=1:e-1;
52      plot(x,errT(x),x,errV(x));
53      xlabel('Epoche ') % x-axis label
54      ylabel('Valori di errore ') % y-axis label
55      legend('Errore Training Set','Errore Validation Set')
56
57  end
```

Riferimenti bibliografici

[1] Martin Riedmiller, *Rprop - Description and Implementation Details*, Citeseer, 1994.

[2] Christopher Bishop, *Neural networks for pattern recognition*, Clarendon Press, 1996.

[3] Roberto Prevete, *Appunti del corso di Machine Learning Mod. B.*